Chapter 2 - Vue Template Syntax

Asst.Prof. Dr. Umaporn Supasitthimethee
ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://vuejs.org/guide/essentials/template-syntax.html#template-syntax

Vue Two Core Features

- **Declarative Rendering**: Vue extends standard HTML with templating syntax that allows us to declaratively describe HTML output based on JavaScript state.
- **Reactivity**: Vue automatically tracks JavaScript state changes and efficiently updates the DOM when changes happen.

HTML-based Template Syntax

- Vue.js uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying component instance's data.
- All Vue.js templates are valid HTML that can be parsed by spec-compliant browsers and HTML parsers.
- Under the hood, Vue compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.

Template Syntax

- Directives are special attributes with the $v-$ prefix.
  - `v-text`
  - `v-html`
  - `v-bind`
  - `v-show`
  - `v-if`
  - `v-else`
  - `v-else-if`
  - `v-for`
  - `v-model`
  - `v-on`

Template Syntax : `v-text`

- `v-text` or using `{{ }}` "Mustache" syntax (double curly braces)
- update the **element’s textContent**
- The mustache tag will be replaced with the value of the `counter` property from the corresponding component instance.

```
<script setup>
const counter = 1
</script>

<template>
  Counter:
// <p v-text="counter"></p>
<p>{{ counter }}</p>

</template>

<style></style>
```

Template Syntax : `v-html`

- The double mustaches interpret the data as plain text, not HTML.
- In order to output real HTML, you will need to use the `v-html` directive:

```
<script setup>
const rawHTML = '<b><i>This should be bold and italic</i></b>'
const notHTML = '<b><i>This should not be bold and italic</i></b>'
</script>

<template>
  <p v-html="rawHTML"></p>
  <p>{{ notHTML }}</p>
  </template>

<style></style>
```

Template Syntax : `v-bind`

- Mustaches cannot be used inside HTML attributes. Instead, use a `v-bind` directive:
- `v-bind` or using colon (:) for short syntax
- Dynamically bind **one or more attributes**, or a **component prop** to an expression.

```
<script setup>
const headingStyle = 'color:Green'
</script>

<template>
    <!-- <h2 :style="headingStyle">Hello, Vue</h2> -->
    <h2 v-bind:style="headingStyle">Hello, Vue</h2>

</template>

<style></style>
```

Boolean Attributes

- Boolean attributes are attributes that can indicate `true/false` values by their presence on an element. For example, `disabled` is one of the most commonly used boolean attributes.
- `v-bind` works a bit differently in this case:
- The disabled attribute will be included if `isButtonDisabled` has a truthy value.
- It will also be included if the value is an empty string, maintaining consistency with `<button disabled="">`.
- For other falsy values the attribute will be omitted.

```
<button :disabled="isButtonDisabled">Button</button>
```

Using JavaScript Expressions

- Vue supports the full power of JavaScript expressions inside all data bindings:

```
{{ number + 1 }}
{{ ok ? 'YES' : 'NO' }}
{{ message.split('').reverse().join('') }}
 <div :id="`list-${id}`"></div>
```

- In Vue templates, JavaScript expressions can be used in the following positions:
  - Inside text interpolations (mustaches)
  - In the attribute value of any Vue directives (special attributes that start with v-)
- Each binding can only contain **one single expression**. An expression is a piece of code that can be evaluated to a value.

Expressions Only

Therefore, the following will **NOT** work:

```
<!-- this is a statement, not an expression: -->
{{ var a = 1 }}

<!-- flow control won't work either, use ternary expressions -->
{{ if (ok) { return message } }}
```

Calling Functions

- It is possible to call a component-exposed method inside a binding expression:

```
<span :title="toTitleDate(date)">
  {{ formatDate(date) }}
</span>
```

Template Syntax : `v-show`

- Toggles the element's `display` CSS property based on the truthy-ness of the expression value.
- An element with `v-show` will always be rendered and remain in the DOM (using `style: "display: none"`)
- This directive triggers transitions when its condition changes.

```
<script setup>
const notHTML = '<b><i>This should not be 
bold and italic</i></b>’
const hide = false
</script>

<template>
   <p v-show="hide">{{ notHTML }}</p>
</template>

<style></style>
```

Template Syntax : `v-if/v-else/v-else-if`

- The directive `v-if` is used to conditionally render a block. The block will only be rendered if the directive's expression returns a truthy value.
- The `v-else` uses to indicate an "else block" for v-if
- The `v-else-if`, as the name suggests, serves as an "else if block" for v-if. It can also be chained multiple times.

```
<script setup>
const score = 70
</script>
<template>
  <p v-if="score >= 80">Very Good</p>
  <p v-else-if="score >= 70">Good</p>
  <p v-else>Need Improvement</p>

</template>

<style></style>
```

`v-if` **VS.** `v-show`

- `v-if` is "real" conditional rendering because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.
- `v-if` is also lazy: if the condition is false on initial render, it will not do anything - the conditional block won't be rendered until the condition becomes true for the first time.
- In comparison, `v-show` is much simpler - the element is always rendered regardless of initial condition, with CSS-based toggling.
- Generally speaking, `v-if` has higher toggle costs while `v-show` has higher initial render costs.

Usage
- prefer `v-show` if you need to toggle something very often,
- prefer `v-if` if the condition is unlikely to change at runtime.

Template Syntax : `v-for` (with a Range)

- v-for can also take an integer. In this case it will repeat the template that many times, based on a range of 1...n.

```
<span v-for="n in 10">{{ n }}</span>
```

Template Syntax : `v-for` (for Array)

- We can use the `v-for` directive to render a list of items based on an array.
- The `v-for` directive requires a special syntax in the form of `course in courses`, where `courses` is the source data array and `course` is an alias for the array element being iterated on:

```
<script setup>
const courses = [
  'Discrete Mathematics for Information Technology',
  'Statistics for Information Technology',
  'Applied Mathematic for data science',
  'Programming Fundamental'
]
</script>
<template>
   <ul >
      <li v-for="course in courses">{{ course }}</li>
    </ul>
</template>
<style></style>
```

Maintaining State

- To give Vue a hint so that it can track each node's identity, and thus reuse and reorder existing elements, you need to provide a unique key attribute for each item:
- It is recommended to provide a `key` attribute with `v-for` whenever possible

```
<ul>
    <li v-for="(course, index) in filterCourses" :key= "course.id">
      {{ index + 1 }}- {{ course }}
    </li>
  </ul>
```

Template Syntax : `v-for` (for Object)

You can also use `v-for` to iterate through the properties of an object.

```
<script setup>
const authors = {
  firstname: 'Evan',
  lastname: 'You'
 }
</script>
<template>
  <div>
    <p v-for="(value, key, index) in authors" :key="index">
      {{ index }}-{{ key }}: {{ value }}
    </p>
  </div>
</template>
<style></style>
```

Use Destructuring on the `v-for`

```
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

```
<li v-for="{ message } in items"> {{ message }} </li>
<!-- with index alias -->
<li v-for="({ message }, index) in items"> {{ message }} {{ index }} </li>
```

Nested `v-for`

```
<li v-for="item in items">
  <span v-for="childItem in item.children">
    {{ item.message }} {{ childItem }}
  </span>
</li>
```

Reactivity Variable with `ref()`

- Vue provides a `ref()` function which allows us to create reactive that can hold any value type.
- `ref()` takes the argument and returns it wrapped within **a ref object** with a `.value` property.
- When `ref` are accessed as top-level properties in the template, they are automatically "unwrapped" so there is no need to use `.value`

```
<script setup>
import { ref } from 'vue'
const msg = ref('hello, vue')
setTimeout(() => {
  msg.value = 'good bye'
  console.log(msg)
}, 3000)
</script>

<template>
  <p>Message: {{ msg }}</p>
</template>

<style></style>
```

Using `ref()` with Object and Array

//ref() with Object

```
<script setup>
import { ref} from 'vue'
const profile = ref({
  name: 'Evan You',
  creator: 'Vue.js'
})
console.log(profile.value.name)
console.log(profile.value.creator)

</script>
<template>
  <p>{{ profile.name }}</p>
  <p>{{ profile.creator }}</p>
</template>
<style></style>
```

//ref() with Array

```
<script setup>
import { ref } from 'vue'
const topics = ref(['template syntax',
'declarative rendering', 'ref function'])
console.log(topics.value[0])
console.log(topics.value[1])
console.log(topics.value[2])
// topics.value.forEach((topic) =>
console.log(topic))

</script>
<template>
  <div>{{ topics[0] }}</div>
  <div>{{ topics[1] }}</div>
  <div>{{ topics[2] }}</div>
</template>
<style></style>
```

Reactivity Variable with `reactive()`

- We can create a reactive object or array with the `reactive()` function.
- It only works for **object types** (objects, arrays, and collection types such as Map and Set).
- It **cannot** hold primitive types such as string, number or Boolean.

Using `reactive()` with Object and Array

//ref() with Object

```
<script setup>
import { reactive} from 'vue'
const profile = reactive({
  name: 'Evan You',
  creator: 'Vue.js'
})
console.log(profile.name)
console.log(profile.creator)

</script>
<template>
  <p>{{ profile.name }}</p>
  <p>{{ profile.creator }}</p>
</template>
<style></style>
```

//ref() with Array

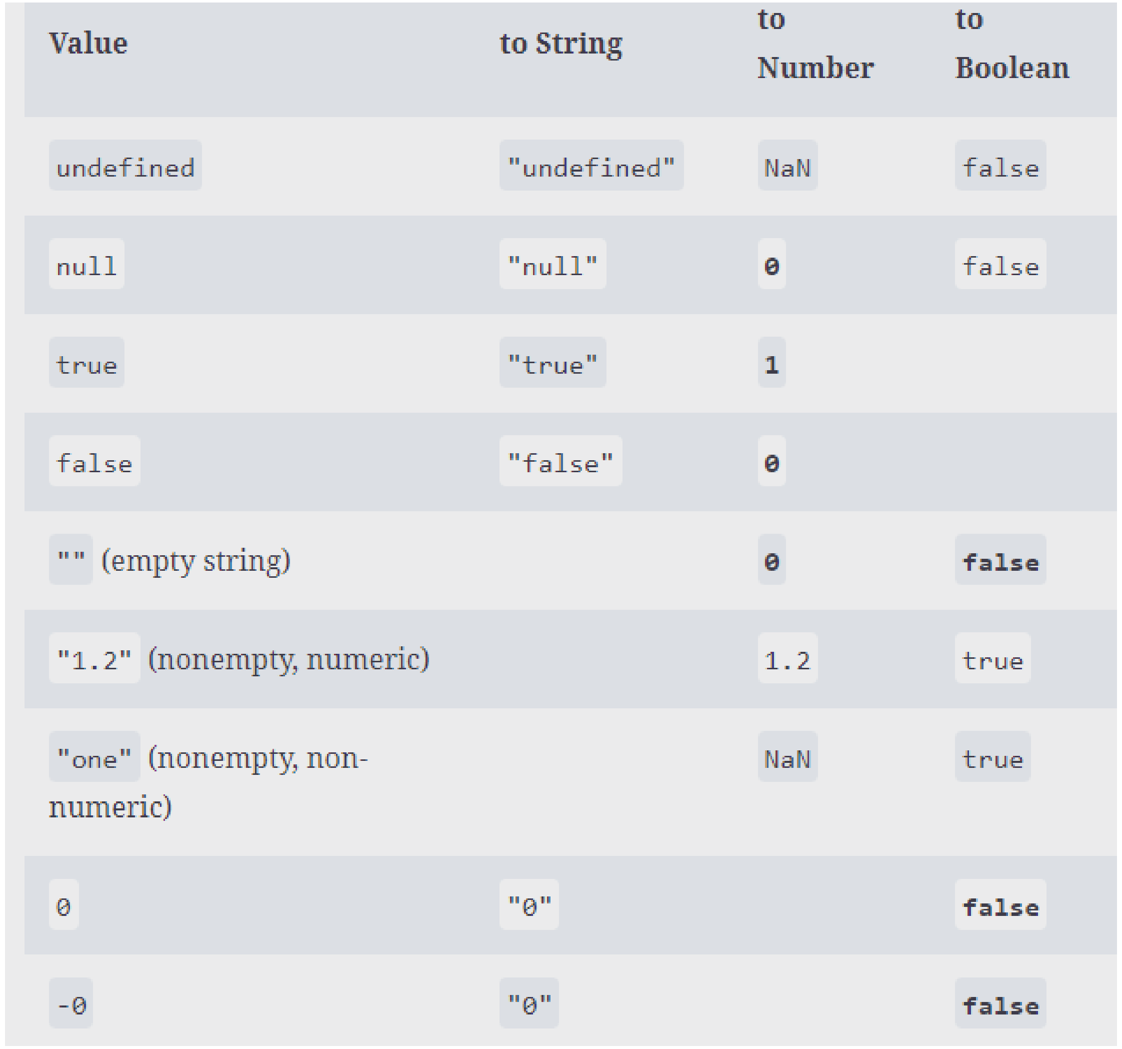
```
<script setup>
import { reactive } from 'vue'
const topics = reactive(['template syntax',
'declarative rendering', 'ref function’])
console.log(topics[0])
console.log(topics[1])
console.log(topics[2])
// topics.forEach((topic) => console.log(topic))
</script>
<template>
  <div>{{ topics[0] }}</div>
  <div>{{ topics[1] }}</div>
  <div>{{ topics[2] }}</div>
</template>
<style></style>
```

Reactive() to Ref() Function

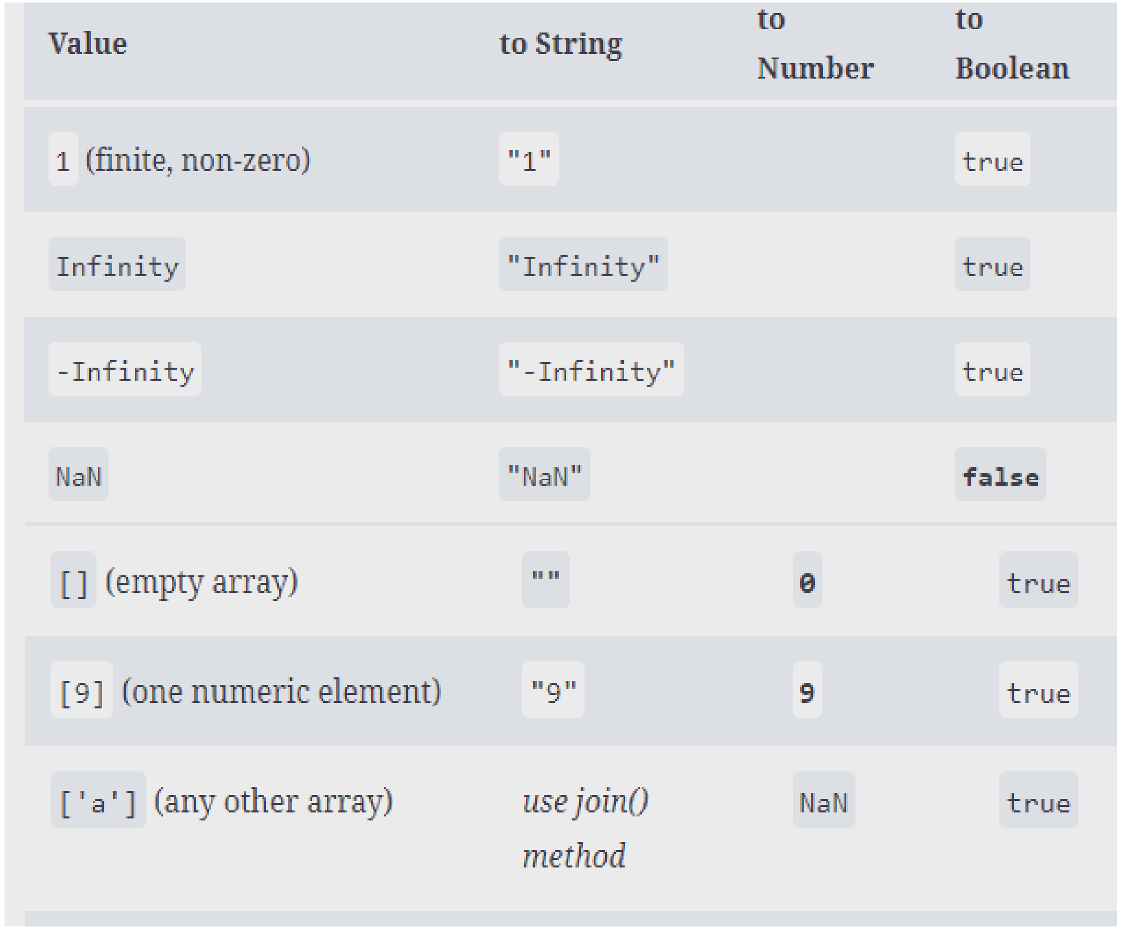
```
<script setup>
import { ref, reactive, toRef, toRefs } from 'vue'
const cube = reactive({
  length: 10,
  width: 20,
  height: 33
})
const length = toRef(cube, 'length')
console.log(length.value)
console.log(cube.width)

const { width, height } = toRefs(cube)
console.log(width.value)
console.log(height.value)
</script>
<template>
  <p>{{ length }}</p>
  <p>{{ width }}</p>
  <p>{{ height }}</p>
</template>
```

JavaScript implicit type conversions



Value	to String	to Number	to Boolean
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	`0`	`false`
`true`	`"true"`	`1`	
`false`	`"false"`	`0`	
`""` (empty string)		`0`	`false`
`"1.2"` (nonempty, numeric)		`1.2`	`true`
`"one"` (nonempty, non-numeric)		`NaN`	`true`
`0`	`"0"`		`false`
`-0`	`"0"`		`false`



Value	to String	to Number	to Boolean
`1` (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		`false`
`[]` (empty array)	`""`	`0`	`true`
`[9]` (one numeric element)	`"9"`	`9`	`true`
`['a']` (any other array)	*use join() method*	`NaN`	`true`